APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: LOGIC SIMULATION

APPLICANT: WILLIAM R. WHEELER, TIMOTHY J. FENNEL, AND MATTHEW J. ADILETTA

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL485673417US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

August 29, 2001
Date of Deposit

Samantha Bell
Signature

Samantha Bell
Typed or Printed Name of Person Signing Certificate

LOGIC SIMULATION

BACKGROUND

[0001] This invention relates to logic simulation.

[0002] Logic designs for computer chips typically include combinatorial elements and state elements. Combinatorial elements, such as AND gates and OR gates, combine two or more logic states to produce an output. State elements, such as latches and flip-flops (FFs), hold a logic state for a period of time, usually until receipt of an external clock signal.

[0003] Computer languages exist which allow designers to simulate logic designs, including combinatorial and state elements, prior to forming the logic on silicon. Examples of such languages include Verilog and Very High-Level Design Language (VHDL). Using these languages, a designer can write code to simulate a logic design and execute the code in order to determine if the logic design performs properly.

[0004] In simulating the logic design, the code can perform two state simulation and/or four state simulation. In two state simulation, the code models two logic states: one and zero. In four state simulation, the code models the one and zero logic states handled during two state simulation, an undefined state, and a high impedance state. Two state simulation can quickly execute and detect errors in high and low logic states in the logic design while four state

simulation can detect design problems when signals in the logic design are not initialized (e.g., are undefined) or driven by other logic (e.g., have high impedance).

[0005] Standard computer languages may also be used to simulate a logic design. One example of a standard computer language that may be used is C++.

DESCRIPTION OF DRAWINGS

[0006] FIG. 1 is a flowchart showing a process of simulating operation of a logic design.

[0007] FIG. 2 is a block diagram of a logic design containing individual state and combinatorial elements.

[0008] FIG. 3 is a block diagram of an alternative logic design containing state and combinatorial elements.

[0009] FIG. 4 is a flowchart showing a process of performing three state simulation initialization checks.

[0010] FIG. 5 is a flowchart showing a process of handling forced nodes in cycle-based simulation.

[0011] FIG. 6 is a block diagram of a computer system on which the processes of FIGS. 1, 4, and/or 5 may be performed.

DESCRIPTION

[0012] Referring to FIG. 1, a process 100 illustrates an example of simulating operation of a hardware circuit or logic design represented by a block diagram using three state

simulation. The process 100 may be implemented using a computer program running on a computer or other type of machine as described in more detail below.

[0013] By using three state simulation, the process 100 can model a state at a node in the block diagram as a logic high (e.g., a one), as a logic low (e.g., a zero), or as an undefined value. (A logic high may be a zero if a logic low is a one.) In this way, the process 100 can perform simulation as in a traditional two state simulation with logic high and low states while detecting undefined states (improperly reset or improperly initialized designs) quickly, accurately, and early in the logic design and simulation process. Upon detection of an undefined state, the logic designer can change the logic design so as to get the state properly initialized, and so forth.

[0014] Each block in the block diagram may represent individual elements or combinations of elements. For example, FIG. 2 shows a graphical representation of a logic design 200 containing combinatorial logic elements 202, 204, 206, and 208 and state logic elements 210 and 212. In the logic design 200, each block represents a single combinatorial element (e.g., multiplexors 206 and 208) or state elements (e.g., FFs 210 and 212). By contrast, in logic design 300 (FIG. 3), the functionality of several combinatorial elements is contained

in a single combinatorial block 302 and the function of several state elements is contained in a single state block 304.

[0015] Once the graphical representation of the logic design has been completed 102 (e.g., FIGS. 2 and 3), the process 100 performs 104 an error check on the design to determine if there are any problems with the design. For example, the process 100 may determine if there are any unterminated or inconsistent connections in the design. If any such problems are detected, the process 100 can issue an error message to a logic designer. The error message may specify the nature of the problem and its location within the logic design. The logic designer is then given the opportunity to correct the problem before the process 100 moves forward.

[0016] The process 100 writes 106 a design wirelist in computer code that simulates the logic design with each of the graphic elements.

[0017] Referring to FIG. 3, for example, the process 100 associates computer code with the combinatorial logic element 302 to define its function and with state logic element 304 to define its function. Furthermore, the process 100 knows which nodes are subject to three state simulation processing (those nodes between logic elements) and tags those nodes as

tri-state nodes in the computer code. The same associating and tagging is true for the logic elements of FIG. 2. The computer code can be generated using any generation process.

[0018] For example, the process 100 may include receiving intermediate computer code (e.g., an application-specific code such as C++ or Verilog) for each graphic element written by the logic designer. Assuming that there are no problems in the design or that the problems have been corrected, the process 100 may generate standard simulation code for the design in any kind of computer code, such as C++, Verilog, VHDL, and other similar codes.

[0019] Once the process 100 has computer code that simulates the logic design, the process 100 performs 108 three state simulation initialization checks for the logic elements included in the logic design. In executing the code in the simulation, the process 100 checks each tri-state signal (those signals on wires that connect tri-state drivers as tagged in the code) to make sure that two (or more) drivers are not driving at the same time on any bit of a tri-state signal. The process 100 checks every bit position of the tri-state signal, e.g., each wire driving a logic device, to make sure that at least one tri-state driver is enabled, e.g., that a signal exists on the wire. Any bit of the signal that

is not driven by one driver is flagged as not being driven or as multiply driven.

[0020] Referring to FIG. 4, a tri-state process 400 illustrates an example of how the process 100 may perform such three state simulation initialization checks. The tri-state process 400 begins 402 by executing the computer code starting at the beginning of the code. In other words, the simulation is performed in a cycle-based, code-ordered fashion as a single call model, running through the code once and establishing all states in the coded logic design.

[0021] The tri-state process 400 may store state information in a database. If the code is in a language with inheritance capabilities such as C++, the tri-state process 400 can enable the code compiler to handle the large number of states that may result from a given logic model. That is, the state of an initial logic gate may be defined as a class (e.g., a C++ class). The states of the logic gates that depend from the initial logic gate may refer back to the state of the initial logic gate without actually including data for the state of the initial logic gate. This way, if the state of a subsequent gate depends on the state of a preceding gate, it is possible to obtain the state of the preceding gate without actually adding more data to the database.

[0022] When the tri-state process 400 encounters 404 a tri-state signal in the code, the tri-state process 400 determines 406 if two (or more) drivers are driving any bit of the signal. If so, then a potential contention exists on the wire line, so the output of the tri-state signal has an undefined state. The tri-state process 400 sets 408 the undefined bit for that signal, indicating an undefined state for that signal. The tri-state process 400 continues 410 executing the code. Alternatively, the tri-state process 400 may terminate or suspend the simulation upon detection of the undefined state and issue 412 an error message to the logic designer as described above. The logic designer fixes the problem and the tri-state process 400 can resume simulation of the logic design at the point of termination or at the beginning of the code.

[0023] If two (or more) drivers do not drive the signal, then the tri-state process 400 determines 414 if any source drives the signal. If not, then the tri-state process 400 can deduce on the fly that the signal has a high impedance state. The tri-state process 400 may not hold information in a bit on the high impedance state, but, if it does, it sets 416 a high-impedance bit for that signal. Whether the tri-state process 400 sets a high impedance bit or not, the tri-state

process 400 may continue 410 executing the code or may issue 412 an error message as described above.

[0024] If less than two drivers but more than zero drivers drive the signal, then one driver drives the signal. The tri-state process 400 sets 418 the appropriate one of the logic high and logic low states for the signal, logic high for a signal state of one and logic low for a signal state of zero. The tri-state process 400 continues 410 executing the code as described above.

[0025] Referring again to FIG. 1, after performing the three state simulation initialization checks, the process 100 determines 110 if the checks were successful. The checks are successful if the process 100 does not detect any undefined states. If the checks were unsuccessful, then the process 100 ends until another design is entered 102, e.g., until the logic designer modifies the logic design to try to fix the detected state initialization error or errors. (If the process 100 terminates the checks upon detecting an undefined state and/or a high impedance state, then the process 100 knows that the checks were not successful and ends until another design is entered.)

[0026] If the checks were successful, then the process 100 may perform 112 another check or checks, such as four state simulation. Four state simulation typically involves

simulating the logic design using four bits to represent four logic states: logic high, logic low, undefined, and high impedance. Performing four state simulation can help refine the initialization model deemed successful under the previously performed three state simulation. The process 100 need not perform the four state simulation or any other checks after three state simulation.

[0027] The process 100 can also determine 114 if the four state simulation was successful. In other words, the process 100 determines if any states are tagged as undefined and/or as high impedance. This check can be performed, for example, after the four state simulation. If the checks were unsuccessful, then the process 100 ends until another design is entered as described above. If the checks were successful, then the process ends 116, indicating that the logic design is properly initialized.

[0028] In executing the code during simulation, the process 100 performs logic computations, e.g., AND operations for AND gates, and attempts to store results of the logic computations in memory. Memory is typically divided into pages (sometimes called virtual memory pages), each page including a defined amount of data (usually expressed in bytes) corresponding to code. If a page is write-protected, then the process 100 cannot write the result to memory, resulting in an error.

[0029] The process 100 may use the fact that pages can be write-protected to handle forced results. A result is forced if the logic designer, typically for purposes of simulation, defines the result to be a particular value regardless of the actual result of a logic computation.

[0030] While executing the code, the process 100 could check each node to determine if the node has a forced value. This determination, however, can slow down the simulation and use valuable processing resources. Instead, the process 100 may identify forced nodes while writing the design wirelist and set access rights for the page or pages including those forced nodes as write-protected. Then, when the process 100 executes the code, forced nodes may be identified by an attempt to write to a write-protected memory page rather than by a slower node-by-node evaluation of forced values.

[0031] Referring to FIG. 5, a force process 500 illustrates an example of how the process 100 may handle storage of logic results and forced nodes during cycle-based simulation. In generating the code for simulation, the memory page or pages associated with any forced nodes included in the logic design are write-protected. In this way, the force process 500 can identify which nodes are forced by encountering storage errors as further described below. The force process 500 may be used in any cycle-based simulation.

[0032] The force process 500 may perform 502 a logic computation and attempt 504 to store the result of the logic computation in memory. If storage cannot occur for some reason, the force process 500 generates an exception indicating an error condition. If the force process 500 does not generate an exception in attempting to store the result in the memory page, then the force process 500 stores 506 the result in memory and continues 508 executing the code.

[0033] If the force process 500 does generate an exception, then the force process 500 calls or executes an exception handler to interrogate the system to determine the reason for the exception. A plurality of exception handlers may be available to the force process 500, each exception handler capable of handling a particular type of exception, e.g., write access errors, invalid memory address, etc. Typically, exception handlers are accessed in a predetermined order, where the exception is passed from one exception handler to the next until the exception reaches the exception handler capable of handling that particular type of exception. In this example, the force process 500 first passes the exception to a first exception handler capable of handling write access errors.

[0034] The first exception handler determines 510 if the exception was generated because of a write access problem. If

not, then the first exception handler passes 512 the exception to the next exception handler according to the predetermined order. This exception handler either handles the exception or passes the exception to the next exception handler, and so on until the exception reaches the appropriate exception handler and is handled.

[0035] If the error was generated because of a write access problem, then the first exception handler handles the exception. The first exception handler knows which instruction in the code caused the exception because the instruction that caused the force process 500 to generate the exception is usually included in the exception. The first exception handler stores 514 this instruction, as yet unexecuted, at the top of a non-write protected memory page. This memory page is typically a memory page that does not include any of the code being executed by the process 100 and may be a page set aside for use by the first exception handler. Immediately following this instruction in the memory page, the first exception handler inserts 516 an illegal instruction that will always generate an exception when executed. The first exception handler also unprotects 518 the original write-protected memory page where the force process 500 tried to store the result.

[0036] The force process 500 then redirects code execution to start execution 520 at the top of the memory page used by the first exception handler. The originally failed instruction executes and because the original memory page is no longer write-protected, the force process 500 can write the result to the original memory page. In this way, the force process 500 can execute the instruction as if the artificially forced value was not present and therefore simulate operation of the logic design as it would actually be executed in hardware to increase the chances of finding initialization errors during this simulation.

[0037] The memory page continues executing with the illegal instruction, which generates an exception. This exception passes to an exception handler that determines 522 if the force process 500 wrote to the original memory page.

[0038] If the force process 500 did store the result in the original memory page, then the force process 500 concludes that the value at that node is forced because the page was write-protected, indicating that at least one value stored in that page is forced, and because the result changed from that stored value. Thus, the force process 500 rewrites 524 the original value (the presumably forced value) back into the original memory page.

[0039] The force process 500 also re-protects 526 the original memory page so as to catch any later attempts to overwrite a forced value stored in the original memory page.

[0040] After re-protecting the original memory page, the force process 500 starts 528 executing code at the original memory page at the instruction following the instruction that generated the original exception.

[0041] If the force process 500 did not store the result in the original memory page, then the force process 500 concludes that the value at that node is not forced. In other words, an exception was properly generated because the force process 500 attempted to write the result into a write-protected memory page, but the page was write-protected for a reason other than because the result at that node is forced, e.g., another node value mapped to that page is forced. The force process 500 thus leaves the page as is, re-protects 526 the original memory page, and starts 528 executing code at the original memory page at the instruction following the instruction that generated the original exception.

[0042] Thus, in this scenario where the exception was properly generated but the value was determined to not be forced, the force process 500 performed some ultimately unnecessary operations, particularly those involving the additional memory page. However, as memory pages are usually

quite small (on the order a few kilobytes) and as not many nodes are forced in logic designs (if any are at all), the computation in the force process 500 is typically much less than the computation involved in checking every node for a forced value.

[0043] FIG. 6 shows a computer 600 for performing simulations using the process 100, the tri-state process 400, and the force process 500. Computer 602 includes a processor 604, a memory 606, and a storage medium 608 (e.g., a hard disk). Storage medium 608 stores data 610 which defines a logic design, a graphics library 612 for implementing the logic design, intermediate code 614 and simulation code 616 that represents the logic design, logic simulator programs 618 (e.g., event-driven and/or cycle-based), and machine-executable instructions 620, which are executed by processor 604 out of memory 606 to perform the process 100, the tri-state process 400, and the force process 500 on data 610.

[0044] The process 100, the tri-state process 400, and the force process 500, however, are not limited to use with the hardware and software of FIG. 6; they each may find applicability in any computing or processing environment. The process 100, the tri-state process 400, and/or the force process 500 may be implemented in hardware, software, or a

combination of the two. The process 100, the tri-state process 400, and/or the force process 500 may be implemented in computer programs executing on programmable computers or other machines that each includes a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and one or more output devices. Program code may be applied to data entered using an input device, such as a mouse or a keyboard, to perform the process 100, the tri-state process 400, and/or the force process 500 and to generate a simulation.

[0045] Each such program may be implemented in a high level procedural or object-oriented programming language to communicate with a computer system. However, the programs can be implemented in assembly or machine language. The language may be a compiled or an interpreted language.

[0046] Each computer program may be stored on an article of manufacture, such as a storage medium or device (e.g., CD-ROM, hard disk, or magnetic diskette), that is readable by a general or special purpose programmable machine for configuring and operating the machine when the storage medium or device is read by the machine to perform the process 100. The process 100, the tri-state process 400, and the force process 500 may also be implemented as a machine-readable

storage medium, configured with a computer program, where, upon execution, instructions in the computer program cause the machine to operate in accordance with the process 100, the tri-state process 400, and the force process 500.

[0047] The invention is not limited to the specific embodiments set forth above. For example, the process 100 is not limited to simulating only combinatorial and state logic elements. Other logic elements may be simulated. The process 100, the tri-state process 400, and the force process 500 are not limited to the computer languages set forth above, e.g., Verilog, C++, and VHDL. It may be implemented using any appropriate computer language. Elements of the processes presented may be executed in a different order than that shown to produce an acceptable result.

[0048] Other embodiments not described herein are also within the scope of the following claims.